On the Avoidance of Fruitless Wraps in Grammatical Evolution

Conor Ryan[1], Maarten Keijzer[2], and Miguel Nicolau[1]

[1] Department Of Computer Science And Information Systems
University of Limerick, Ireland
{Conor.Ryan,Miguel.Nicolau}@ul.ie
[2] CS Dept., Free University, Amsterdam
mkeijzer@cs.vu.nl

Abstract. Grammatical Evolution (GE) is an evolutionary system that employs variable length linear chromosomes to represent computer programs. GE uses the individuals to produce derivation trees that adhere to a Backus Naur Form grammar, which are then mapped onto a program. One unusual characteristic of the system is the manner in which chromosomes can be "wrapped", that is, if an individual has used up all of its genes before a program is completely mapped, the chromosome is reread. While this doesn't guarantee that an individual will map, prior work suggested that wrapping is beneficial for the system, both in terms of increased success rates and a reduced number of invalid individuals. However, there has been no research into the number of times an individual should be wrapped before the system gives up, and an arbitrary upper limit is usually chosen.
This paper discusses the different types of grammars that could be used with this system, and indicates the circumstances under which individuals will fail. It then presents a heuristic to minimize the number of wraps that have to be made before the system can determine that an individual will fail. It is shown that this can drastically reduce the amount of wrapping on a pathologically difficult problem, as well as on two *classes* of grammar often used by the system.

1 Introduction

Grammatical Evolution(GE) [10][6] is an Evolutionary Automatic Programming system that uses a variable length Genetic Algorithm to evolve programs in any language. The key to the system is the manner in which a Backus Naur Form (BNF) grammar is employed to specify the target language, and is used to map the linear genomes into syntactically correct programs.

One unusual characteristic of GE is the manner in which an individual's genome is reused, using a technique known as *wrapping*. That is, if, when the end of the genome is reached, an individual hasn't fully mapped to a program, another pass is made through the genome. The *intrinsically polymorphic* [3], that is, the manner in which the meaning of a codon is dependant on the context in which it is used, nature of the codons on the genome means that it is possible

E. Cantú-Paz et al. (Eds.): GECCO 2003, LNCS 2724, pp. 1752–1763, 2003.

that, on the second and subsequent passes, a different interpretation will be produced. This can lead to a complete mapping for an individual that would otherwise have failed.

Although previous work [5] illustrated that wrapping often helps evolution and, at worst, does not harm it, no research has been conducted into the how many times an individual should wrap. An upper limit is chosen, but there is no way to tell in advance how this number should be set. If it is too low, then potentially useful individuals will wrongly be dubbed as invalid, while if it is too high, the system will waste time wrapping individuals that will *never* map.

This paper takes a formal look at the process of wrapping, and devises a heuristic to determine as early as possible when the wrapping of an individual should stop. We show that this heuristic reduces the effort required by GE without effecting the performance, because only those individuals that would fail to wrap are removed from the population.

The paper is laid out as follows. The next section introduces Grammatical Evolution and Backus Naur Form, while Sect. 3 discusses some of the properties of the simplest type of context free grammar(CFG). Section 4 expands this to include CFGs with two non-terminals, and Sect. 5 expands the work further again to the general case. Some experiments are conducted in Sect. 6, before the paper concludes with a brief summary in Sect. 7.

2 Grammatical Evolution

Grammatical Evolution is a genotype-phenotype mapping system that can map variable length binary strings into sentences of an arbitrary language. To use the system, one specifies the target language using a Backus Naur Form (BNF) grammar. BNF is a convenient representation for describing grammars. A grammar is represented by a tuple $\{N, T, P, S\}$, where T is a set of terminal symbols, i.e., items that can appear in legal sentences of the grammar, and N is a set of non-terminal symbols, which are interim items used in the generation of terminals. P is a set of production rules that map the non-terminal symbols to a sequence of terminal (or non-terminal) symbols, and S is a start symbol, from which all legal sentences must be generated.

Below is a sample grammar, similar to that used by Koza [2] in his symbolic regression and integration problems. Although Koza did not use grammars, the terminals in this grammar are similar to his function and terminal sets.

```
S = <expr>
<expr>     ::=  <expr> <op> <expr>
              | ( <expr> <op> <expr>)
              | <pre-op> ( <expr> )
              | <var>
<op>       ::= + | - | / | *
<pre-op>   ::= Sin | Cos | Exp | Log
<var>      ::= 1.0
```

2.1 The Mapping Process

Rather than encoding actual programs, GE encodes *choices*. At many stages during the derivation of a legal sentence, one is required to make a choice. For example, in the previous grammar, there are four possible choices when mapping the `<expr>` non-terminal. To decode the choices, one first looks upon the genome as being divided into 8-bit codons, each of which codes for a single choice. Consider the individual below, written in decimal for brevity.

```
220 203 51 123 2 45
```

Initially, the *goal-stack*, that is, the stack of non-terminals, contains `<expr>`, as this is the start symbol. The first codon is read and **mod**ed by the number of choices giving, in this case, 0, so the first choice is made, causing the top of the stack to be replaced by the newly produced non-terminals. The individual continues to be read, codon by codon, until either the individual has mapped, or the all the codons have been exhausted.

In the latter case, the individual is *wrapped*. That is, the genome is read again. Clearly, if the first non-terminal to be mapped is the same as the *start* symbol, the individual will keep growing, and will never terminate. However, if the non-terminal is different, then this time around, the first codon will be used differently. This change will then ripple down through the rest of the codons, as the meaning of each codon is dependent on the context in which it is used. This property is referred to as *intrinsic polymorphism*. Even with wrapping, however, there is no guarantee that an individual will map to completion, so an upper limit is placed on the number of wraps permitted. Once this number has been reached, an individual is terminated.

Although the choice of this limit is clearly very important, given that if it is too low, individuals may be terminated before they get a chance to map, while an

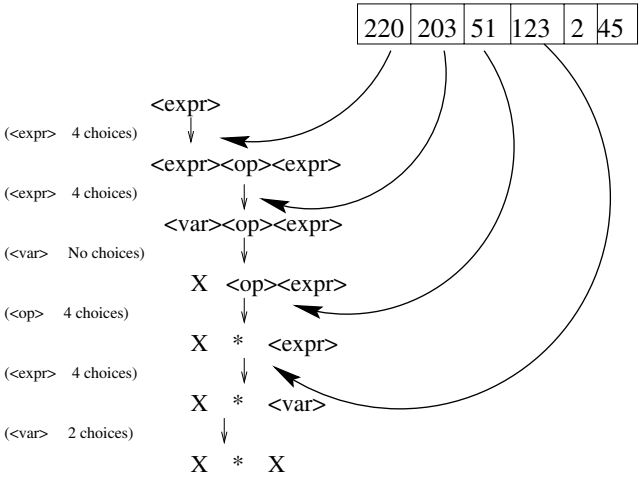

Fig. 1. Example mapping outline

excessively high limit leads to excessive wraps and memory usage (as individuals doomed to failure from excessive wraps tend to keep growing), no investigation has been carried out into determining a suitable number. An upper limit of ten was suggested in [4], although in a related system [9] a different approach is used, in which wrapping is only permitted in the first generation. These individuals are then "unrolled" so that they map in a single pass, and no wrapping is permitted in subsequent generations. In all cases, individuals that fail to map are given zero fitness, and are not permitted to engage in any reproductive activity.

Investigations into wrapping [5] suggested that, in the worst case, it doesn't adversely affect the performance of the system, while in the best case, it improves the performance.

These results suggest that, in general, most of the wrapping occurs early on in a run, and that most individuals (regardless of whether they wrap or not) map to completion after the first few generations. That said, there is clearly a lot of wasted evaluation expended on fruitless wraps. If there was some way to identify individuals that will *never* map without performing a large number of wraps, then the system could expect to enjoy a speed up, without any cost in performance, as none of these individuals contribute their genetic material through crossover.

3 Single Non-terminal Grammars

The simplest type of grammars are single non-terminal grammars. These grammars adhere to the "closure" principle of GP, that is, all functions (non-terminals for GE) can take all other functions and terminals as arguments. This can be illustrated with the following grammar, the productions of which are labelled for later reference :

```
E ::= (+ E E) | (* E E) | (- E E) | (% E E) | x | 1
         0          1        2         3      4   5
```

This grammar produces the same individuals as a GP set up with a function set of { +, *, -, %} (each of which has an arity of 2) and a terminal set of {x, 1}. In this case there is just *one* non-terminal, which ensures that any production can be applied at any time.

Furthermore, as there is only one non-terminal, codons effectively have *absolute* values, as they will never be re-interpreted. Thus, an individual that doesn't successfully map on the first pass will *never* map.

Single non-terminal grammars are, however, still useful in this context, as their simplicity helps us provide some definitions. We define *producers* to be those rules which increase the stack size, by adding one more more non-terminals to the stack, e.g. rules 0-3 above. Similarly, *consumers* are those rules which reduce the stack size, e.g. rules 4 and 5 above. Further, those rules which, after application, leave the size of the stack unchanged are referred to as *neutrals*.

We assign each production rule a **PCN** (Producer/Consumer/Neutral) number. That is, the effect it has on the stack size. From the grammar above, the

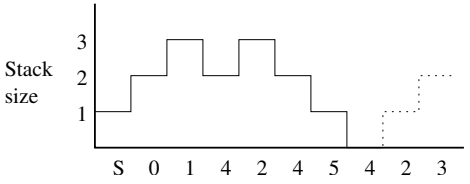

Fig. 2. Example shape graph

first four rules each have a PCN# of 1, while the remaining two have a PCN# of -1. Consider the (decoded) individual **0 1 4 2 4 5 4 2 3**. If we align the codons as follows, we can determine how the stack will grow and contract:

Codon	0	1	4	2	4	5	4	2	3
PCN#	1	1	-1	1	-1	-1	-1	1	1

The stack size always starts at 1, to reflect the presence of the start symbol. As each codon is successively applied, its PCN# is added to the size. If the size reaches zero, the individual has completely mapped.

A useful visualisation method is that of shape graphs [1]. These can be used to indicate the manner in which a stack grows and contracts over time. Figure 2 gives an example of the individual above.

The mapping stops when the size goes to zero, so the unexpressed codons are represented by the dashed line. Because this is a single non-terminal grammar, each codon has an absolute value, so the meaning is not influenced by its predecessors. Thus, we can examine the PCN numbers to determine whether or not an individual will map. That is, the PCN number for string s is

$$pcn(s) = \sum_{i}^{N} pcn(s_i)$$

If this number is less than zero, the mapping process will necessarily terminate. Furthermore, a string will terminate if and only if there is a k such that

$$\sum_{i}^{k} pcn(s_i) = -1$$

and, if a string doesn't terminate the first time around, it never will, as wrapping is essentially a double sum over PCN.

In particular, we can assert that, if the total PCN of consumers is greater than that of producers in an individual, then that individual will map.

Early work [4] indicated that the number of invalid individuals in GE dropped off from an average of 20% in the initial generation to almost zero by the fourth generation. We postulate that this is due to the evolution of a *stop sequence* of codons which, *almost always*, regardless of what productions were performed at the start of the mapping process, will complete an individual. Notice, however,

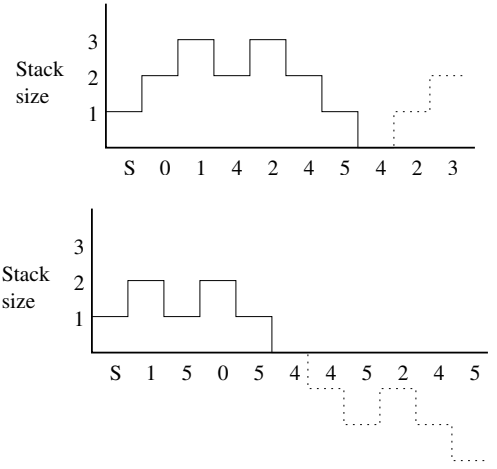

Fig. 3. Example shape graph of two GE individuals, with the second individual containing a particularly strong stop sequence

that one cannot expect a stop sequence to guarantee to complete all individuals, as individuals that grow without bounds initially may not map.

This is illustrated by the shape graphs in Fig. 3, in which the second parent has a tail of unexpressed codons, most of which are consumers. We describe any sequence of codons that, on average, consumes non-terminals as a stop sequence. Clearly, not all stop sequences are equal, with some being stronger than others as they consume more, and so have a steeper slope when plotted on a shape graph. Any individual crossed over with the second parent will be more likely to terminate due to the relatively strong stop sequence in the unexpressed tail of that individual.

4 Dual Non-terminal Grammars

The simplest case in which an individual may usefully be mapped is that in which there are two non-terminals. Consider the simple grammar:

```
A -> aAA | B
B -> a | b
```

where A is the start symbol. Typically, GE will wrap an individual if the stack is non-empty after the first pass. However, if the non-terminal A is on top of the stack, there is no point, as the system has just established that, *for this individual*, applying it to non-terminal A will (a) cause the stack to grow; and (b) leave another A on top of the stack. Clearly, this individual will keep growing.

If, on the other hand, we change context to the other non-terminal, we must go through the individual again. There are three possible results, (a) the individual completes; (b) an A is left on top; and (c) a B is left on top.

If an A is on top, then there is clearly no point in continuing, as the stack will continue growing. On the other hand, if there is a B, our next course of action depends on what happened to the stack. If it increased, or remained the same, then we should stop, while if it decreased, then we should continue wrapping.

Consider the individual 0 0 1, which is expressed using such low numbers to avoid having to perform the **mod** rule. After the first pass, the individual will have mapped to:

```
aaBAA
```

In this case, there is a different non-terminal at the top of the stack, so we wrap the individual to produce :

```
aaaaBAA
```

Although the individual has increased in size due to the introduction of the terminals, the stack has remained the same. Thus, we can assume that this individual will never map.

5 General Case

Most grammars have more than two non-terminals. Unfortunately, in these cases, one cannot be guaranteed that the behaviour of an individual will remain constant even in the same context. Consider the grammar :

```
A -> A | BC
B -> b | b
C -> BBB | BBB
```

where upper case letters denote non-terminals, and lower case letters terminals. If we try to map the individual 01, the following results:

Pass #	Individual
1	BC
2	bBBB
3	bbbB
4	bbbb

The result of pass #2 suggests that, with B on top of the stack, this particular individual will cause the stack to continue to increase in size. However, this is not the case, because, on the following pass, the stack decreases, and mapping is complete on the next pass.

This means that a simple watch on the size of the stack isn't enough to guarantee that an individual is going to fail. In general, an individual that will fail to map will enter into a cyclical behaviour. That is, the individual will

keep being applied to the same non-terminal, or *set* of non-terminals. That is, although the individual may keep changing contexts, there is still a cycle. The difficulty, of course, is spotting these cycles.

Consider the individual above. The non-terminals on top of the stack at the start of each pass (including the first) are A,B,B and B. One could claim that the Bs form a repetitive pattern, but there are enough changes occurring in the stack to prevent it from becoming a pathological pattern. Clearly, only examining the top of the stack doesn't present enough information. We suggest the following heuristic : **if the entire stack from the last pass is at the top of the stack from this pass, then stop.**

This will indicate that the system is stuck in a cycle because, when the same stack from the last time around is on top, the exact same derivation sequence will be repeated. Notice that, while this heuristic isn't guaranteed to stop all wraps, it is guaranteed not to terminate an individual too early.

6 Experiments

To test how well the heuristic performed, an artificial problem was created that actively promotes individuals that wrap. This is achieved by making the fitness function the number of wraps an individual requires to map to completion. An individual that fails to wrap is given a score of zero.

The grammar used in this problem is designed to make it difficult to identify individuals that will fail. In particular, the mutually recursive relationship between the two non-terminals, coupled with the fact that both have productions that produce no non-terminals means that the stack can grow and contract several times throughout a derivation.

The actual grammar used is :

```
A -> BB | a
B -> BB | AA | c
```

Notice that, while this grammar is not designed to solve any particular problem, simply to promote wrapping, this turned out to be a nontrivial task, and individuals that reached the maximum fitness were extremely complicated.

A population size of 500 was run for 100 generations, with steady state tournament selection, and a tournament size of three. All results are averaged over 50 independent runs. The maximum fitness was 99, and the maximum number of wraps allowed was set to 100. Figure 4 shows the average fitness over time, and compares the number of illegal individuals identified by the heuristic to the number that exceeded the maximum number of wraps. In no case did an individual deemed a failure by the heuristic go on to map successfully.

Clearly, the heuristic fails to catch all the individuals that fail to map. However, it does identify an average of 35% of them after just two wraps. This represents a saving of 765,576 wraps for this problem. Given that this grammar was specifically designed to encourage wrapping, this is quite a considerable saving.

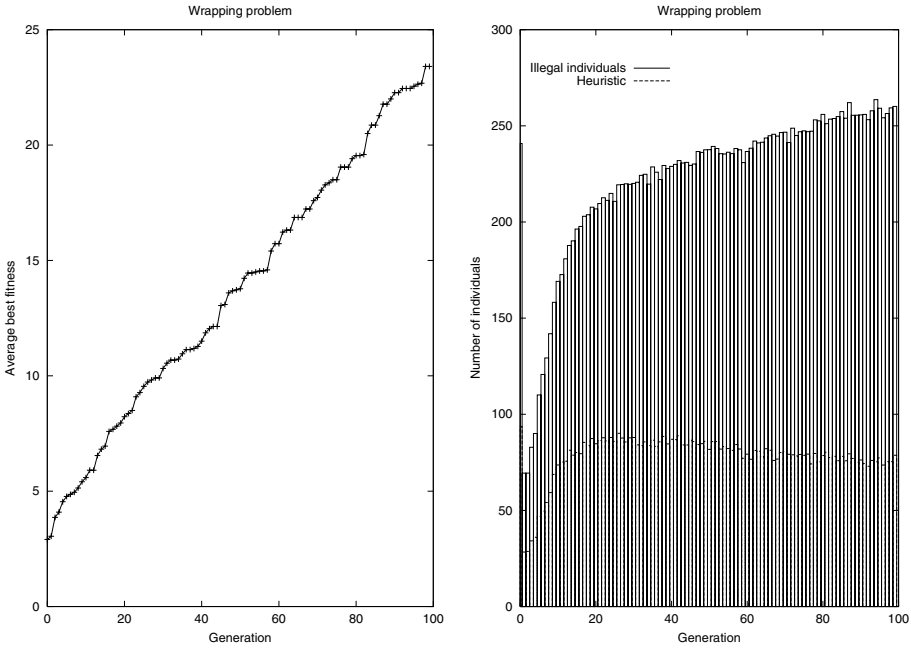

Fig. 4. Average fitness for the artificial wrapping problem (left) and a comparison of the number of individuals flagged by the heuristic, and those that timed out(right)

6.1 Application Grammars

The actual grammars employed by GE do not, in general, have such inherently pathological properties that facilitate wrapping. This section looks at two commonly used grammars, one suited for Symbolic Regression type problems, as shown in Sect. 2, and one for the Santa Fe Ant Trail, as described below.

```
<code> :: =  <line> |<code><line>
<line> :: =  <if-statement>|<op>
<if-statement> :: = if(food_ahead()) {<line>} else {<line>}
<op> :: =    left();  | right();  | move();
```

We are now more interested in the behaviour of the heuristic on a *class* of grammars, rather than on particular problems. To this end, we again use the number of wraps as a fitness function, to create a worst case scenario which may give us a lower bound on how well one can expect the heuristic to perform.

These problems use the same parameters normally associated with GE. That is, a population of 500 evolving for 50 generations, with steady state selection and a mutation probability of 0.01.

Figure 5 shows the performance of each of the grammars using the number of successful wraps as a measure of success. The Santa Fe results immediately leap out, showing that no individual in any of the runs attained a fitness greater

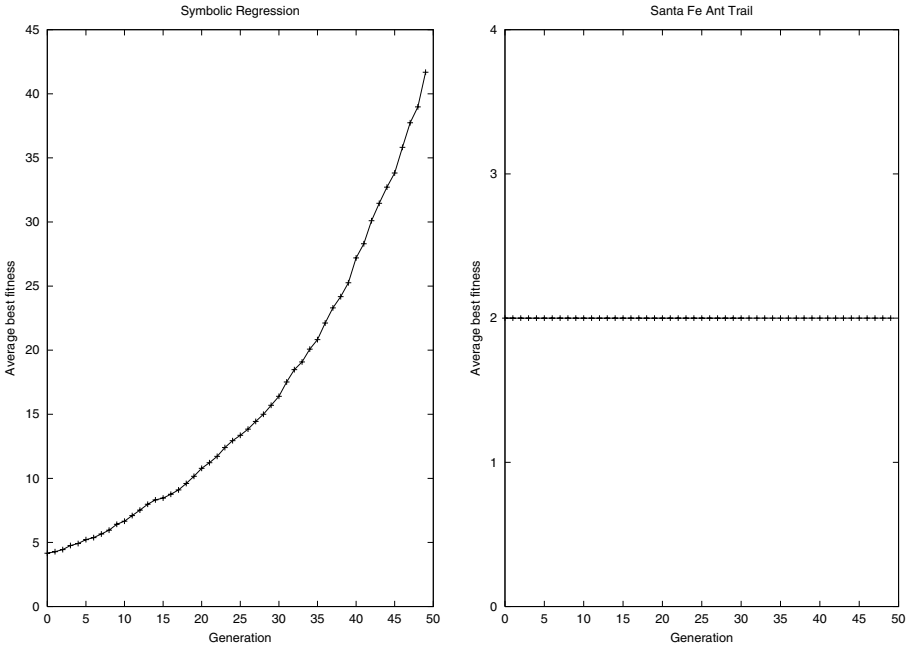

Fig. 5. Performance using number of successful wraps as a measure for Symbolic Regression (left) and Santa Fe Ant (right)

than two. It is an inherent property of the grammar that it will either complete after two wraps or not at all, but the heuristic is still capable of identifying individuals, and does so after an average of 1.3 wraps.

On the other hand, the Symbolic Regression grammar climbs continuously right up until the final generation, clearly indicating that legal individuals in this case are capable of considerably more wraps.

The numbers of individuals identified by the heuristic as being illegal in each case are shown in Fig. 6. In the case of the Symbolic Regression grammar, the heuristic performs very well at the start, but, after about 15 generations, starts to fail to identify failures. Across the 50 generations, the heuristic, on average, identifies 33% of invalid individuals, with an average of 2.06 wraps.

In the case of the Santa Fe grammar, the heuristic performs much better, correctly identifying around 84% of failures, with an average of just 1.3 wraps.

The fact that these problems specifically encourage wrapping should be kept in mind, so, particularly with the symbolic regression grammar, in which the fitness keeps rising, it shouldn't be surprising that the number of invalid individuals keeps increasing. This was done specifically to create a difficult situation for the heuristic, but in usual GE runs, the number of invalid individuals drops to near zero by the fourth or fifth generation.

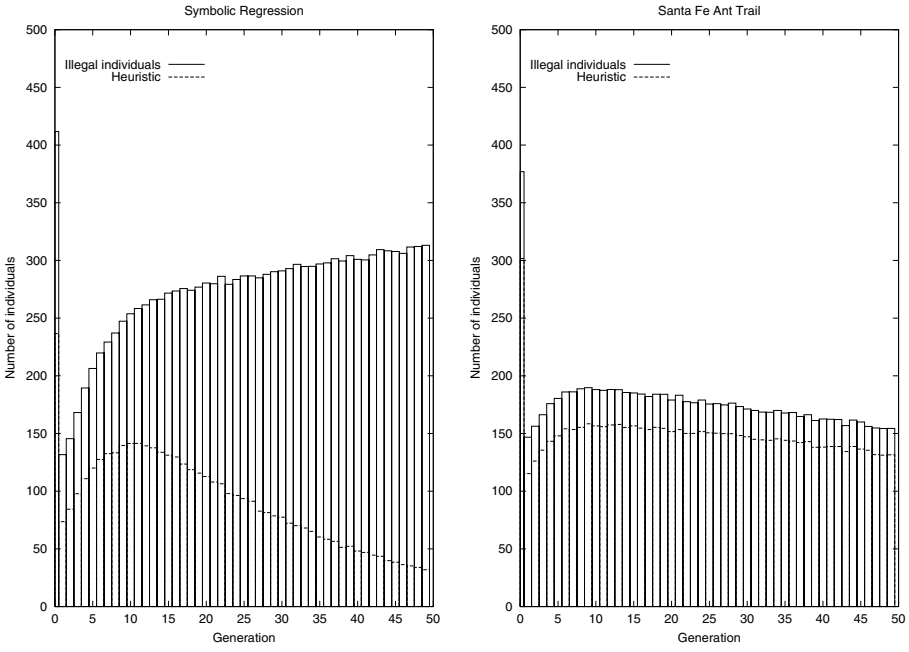

Fig. 6. The number of individuals identified by the heuristic for the Symbolic Regression grammar (left) and the Santa Fe grammar (right)

7 Conclusions

This paper has investigated the phenomenon of wrapping in Grammatical Evolution. We have demonstrated circumstances under which individuals will fail to map using a number of different types of context free grammars, namely, single non-terminal grammars, dual non-terminal grammars and multiple non-terminal grammars. Using shape graphs, we describe a stop sequence, which is any sequence of genes that is likely to terminate another string after crossover.

We showed a simple algorithm to determine whether or not an individual using a dual non-terminal grammar will fail to map, and illustrated the difficulty in extending this algorithm to multiple non-terminal grammars. However, a very cheap heuristic has been shown that, on a problem specifically designed to produce individuals that will fail, successfully identifies 35% of individuals that won't map.

The paper also considered two classes of grammars, a symbolic regression type, and a Santa Fe Ant Trail type. For both of these, the worst case was examined, that is, where individuals are actually rewarded for wrapping. In these cases, the heuristic successfully identified 33% and 84% of failures for the symbolic regression grammar and the Santa Fe Ant Trail grammar respectively.

References

1. Keijzer M. Scientific Discovery using Genetic Programming PhD Thesis, Danish Hydraulic Institute, 2001.
2. Koza, J.R., Genetic Programming: On the Programming of Computers by Means of Natural Evolution, MIT Press, Cambridge, MA, 1992.
3. Keijzer M., Ryan C., O'Neill M., Cattolico M., and Babovic V. Ripple crossover in genetic programming. In *Proceedings of EuroGP 2001*, 2001.
4. M. O'Neill. Automatic Programming in an Arbitrary Language: Evolving Programs with Grammatical Evolution. PhD thesis, University Of Limerick, 2001.
5. O'Neill M. and Ryan C. Genetic code degeneracy: Implications for grammatical evolution and beyond. In *ECAL'99: Proc. of the Fifth European Conference on Artificial Life*, Lausanne, Switzerland, 1999.
6. O'Neill M. and Ryan C. Grammatical Evolution. *IEEE Transactions on Evolutionary Computation*. 2001.
7. O'Sullivan, J., and Ryan, C., An Investigation into the Use of Different Search Strategies with Grammatical Evolution. In the proceedings of *European Conference on Genetic Programming (EuroGP2002)* (pp. 268–277), Springer, 2002.
8. Ryan, C., and Azad, R.M.A., Sensible Initialisation in Chorus. Accepted for *European Conference on Genetic Programming (EuroGP 2003)*.
9. Ryan, C., Azad, A., Sheahan, A., and O'Neill, M., No Coercion and No Prohibition, A Position Independent Encoding Scheme for Evolutionary Algorithms - The Chorus System. In the Proceedings of *European Conference on Genetic Programming (EuroGP 2002)* (pp. 131–141), Springer, 2002.
10. Ryan, C., Collins, J.J., and O'Neill, M., Grammatical Evolution: Evolving Programs for an Arbitrary Language, in *EuroGP'98: Proc. of the First European Workshop on Genetic Programming* (Lecture Notes in Computer Science 1391, pp- 83–95), Springer, Paris, France, 1998.